DAE Request Handling

Problem analysis

Tue, Jan 28, 2003

Java client applications pass their data requests to a DAE node, which acts as a kind of front end's front end. During recent testing of heavy usage of such requests, a problem was noticed in the Linac PowerPC front end nodes that caused the `QMonitor` task to be suspended as a result of a failure while releasing dynamic memory. This note explores possible sources for this bug.

During the testing referred to here, a list of devices was built up, one at a time, that were of interest to the client application, data from which were to be returned at 1 Hz. As each device was added to the list, the DAE revised its request that it made to the front end by canceling its previous request and making a new one with one more device added. The list of devices, here synonymous with device-property pairs, in this case reached 300 or more. There was thus a lot of thrashing of requests being built and canceled. The target front end was almost always `node0600`, which is the server node, or pathway, by which most Linac data is accessed. As a server node, it typically forwards via multicast each request that it initializes so that the other Linac nodes can see it and prepare to respond with their contributions.

The Linac nodes were recently upgraded to operate at 100 Mbps ethernet, which means that network throughput was increased so that more requests could pass through for processing; *i.e.*, the network bandwidth would not throttle the request processing. Consider a `RETDAT` request for 247 nodes, which is about 4000 bytes long and requires three 1500-byte IP fragments to contain it. At 10 Mbps, such a datagram would require about 3.2 ms of bandwidth. At 100 Mbps, only 0.32 ms of bandwidth is required.

In the Linac system code that supports network messaging, there is a scheme for taking care that message blocks are not freed until they are no longer needed. When a message block is queued to the network, a flag is set to indicate that the message block is busy. When a transmission of the datagram containing the message that is contained within the message block has been completed, the `QMonitor` task clears the flag. As long as the flag is set, code that would try to free the message block does not free it, but rather sets another flag bit in the block type field in the block header, signalling that this block is no longer needed by the rest of the system. This flag is noticed by `QMonitor` as it clears the first flag, and on finding this second flag set, `QMonitor` frees the block.

It may be useful to review this logic in seeking out the cause of the bug. For a `RETDAT` request, the message blocks that are used are the forwarded copy of the request and the final reply block for a server request, and the reply block for the nonserver request. To initialize a server request, three blocks are allocated: the request block, the forwarded request message block, and the final reply block. For a nonserver request, three blocks are also allocated: the request block, the internal pointers block, and the reply message block. When a request is canceled, all such blocks are freed.

Consider the nonserver request first, since `node062E` was the one that had its `QMonitor` task suspended. Only the reply block is a message block, so only that one needs the special protection provided via the two flags described above.

For a server request, both the forwarded request block and the final reply block are message blocks, so that both of them need protection from premature releasing.

Within the reply message block is a pointer to the request block. One should be sure that this pointer is not needed or used once a request has been canceled in which the reply block was marked busy.

The OUTPQX routine writes an entry into the OUTPQ (Output Pointer Queue) that points to a message block, advancing the IN offset as it does so. The NetXmit routine concatenates messages as possible using consecutive queue entries and builds datagrams into a transmit buffer, advancing the OUT1 offset. The QMonitor task processes each entry that was used to build the datagram, freeing each mesage block if appropriate and advancing the OUT2 offset. Thus IN moves ahead, OUT1 follows IN, and OUT2 is the last offset to advance. The IN offset is not allowed to advance if doing so would make it equal to OUT2, which would imply the the queue is full. When all three offsets are equal, the queue is empty. (Obviously, if OUT2 catches up to IN, then all three are equal.)

For a server request, the reply block also contains a pointer to the request block. Again, we should not use this pointer if it is possible that the request has already been canceled; *i.e.*, the flag bit in the block type field is set.

Carefully analyze all uses of the NetQFlg field in a message block. Bits in this field tell QMonitor whether or not it should release the memory following completion of transmission of a datagram containing the message in the message block.

The External Request block, which is a message block containing the request message that is to be forwarded, is pointed to by a field in the request block. It can be used to refresh a request to a specific node that has not responded promptly. For RETDAT, this actually means that a new copy of the request is made that contains only devices from that target node, and that selected block can be freed as soon as it has been transmitted.

QMonitor

In each entry in the OUTPQ, a flag is set when the message pointed to by that entry is done. This flag is set in the PowerPC nodes by the Transmitter task, maybe. In the OUTPQMon routine of QMonitor, the OUT2 offset is advanced toward the OUT1 offset, checking in each entry for the flag to be set. As long as each entry checked has the flag set, the FreeBlk routine is called for that message block, and the OUT2 offset is advanced.

FreeBlk uses OUT2 to find the current entry. According to the block type, it clears the busy flag bit. (Different block types use different fields for this purpose.) Then, if the cancel flag bit in the block type field is set, the block is freed. If the cancel flag is not set, further checking is required. For the Acnet message case, which is being studied here, the keep bit is checked in the NetQFlg field, and if it is zero, the block is freed. But if the keep bit is set, this automatic freeing does not take place, and the Acnet answers flag bit is checked. If it is set, and the related request block type is 12, signifying a RETDAT request, the ACDelChk is called. If the related request block type is not 12, then ProHand is called to let the local application handle it, giving it a DelChk calling code. In either case, the request block should be checked to see

whether it is time to cancel the request, say, because it is a one-shot request and the `M` bit in the first word of the Acnet header is zero. (`ACDelChk` looks for the block type = 12 and the `M` bit zero, in which case it callls `ACDelete` to cancel the request. If the canceled bit in the block type field were set, the field would not = 12.) `ACDelete` removes the request from the linked list of active requests, frees the request block, frees any internal ptrs block, and frees the answers block—unless it has the busy bit set in the `NetQFlg` field, in which case it sets the canceled flag bit in the block type field but does not free the answers block. If there is an external request block and a server message id, and the `M` bit is set for multiple replies, it sends a cancel message to the forwarded request destination. The server request message id is released in `LISTP`. The forwarded request block is freed, unless the busy bit in the `NetQFlg` field is set, in which case the canceled bit is set in the block type field.

Server request

After initializing a server request, what is the situation with respect to allocated blocks that support this request? There are 3 blocks:

Block	*`NetQFlg`*	*Meaning*
main request block#12	n.a.	
server request block#9	`0xC000`	busy, keep
final answers block#9	`0x6000`	keep, Acnet answers block

The server request block has the busy bit set in `NetQFlg` because the forwarded message has already been queued to the network. What happens if a cancel message is processed at this point? The main request block and the final answers block can be freed, but the server request block cannot; instead, the flag bit is set in the `MBlkType` field. When `QMonitor` processes the server message block and sees this flag bit, it will free the block.

What about the same question regarding a nonserver request? There are also 3 blocks:

Block	*`NetQFlg`*	*Meaning*
main request block#12	n.a.	
internal ptrs block#14	n.a.	
answers block#9	`0x6000`	keep, Acnet answers block

Because the `0x4000` bit is set in each type#9 Acnet message block, `QMonitor` will not free it. These blocks will stick around until the request is canceled. If a cancel happens at this point, all three blocks will be freed.

Whenever `OUTPQX` is called via `NetQueue` to queue a type#9 message block to the network, The busy bit is set in `NetQFlg`. `QMonitor` clears this bit when it is finished with the block.